

# PL/pgSQL Обзор и конструкции языка



## Авторские права

© Postgres Professional, 2017–2024

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов, Игорь Гнатюк

Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

## Использование материалов курса

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

## Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

[edu@postgrespro.ru](mailto:edu@postgrespro.ru)

## Отказ от ответственности

Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

История PL/pgSQL

Структура блока, объявление переменных

Анонимные блоки

Подпрограммы на языке PL/pgSQL

Условные операторы и циклы

Вычисление выражений

Появился в версии 6.4 в 1998 году

устанавливается по умолчанию с версии 9.0

## Цели создания

простой язык для написания пользовательских функций и триггеров  
добавить управляющие структуры к языку SQL  
сохранить возможность использования любых пользовательских типов, функций и операторов

Родословная: Oracle PL/SQL, Ада

PL/pgSQL — один из первых процедурных языков в PostgreSQL. Он впервые появился в 1998 году в версии 6.4, а с версии 9.0 стал устанавливаться по умолчанию при создании базы данных.

PL/pgSQL дополняет SQL, предоставляя такие возможности процедурных языков, как использование переменных и курсоров, условные операторы, циклы, обработка ошибок и т. д.

PL/pgSQL проектировался на основе языка Oracle PL/SQL, а тот, в свою очередь, был создан на основе подмножества языка Ада. Эта ветвь происходит от таких языков, как Алгол и Паскаль.

Большинство современных языков программирования принадлежат другой ветви — ветви Си-подобных языков, поэтому PL/pgSQL может показаться непривычным, избыточно многословным (характерное отличие — использование ключевых слов BEGIN и END вместо фигурных скобок). Впрочем, этот синтаксис отлично сочетается с синтаксисом SQL.

<https://postgrespro.ru/docs/postgresql/16/plpgsql-overview>

## Метка блока

## Объявления переменных

область действия — блок

область видимости может перекрываться вложенными блоками,  
но можно использовать метку блока

любые типы SQL, ссылки на типы объектов (%TYPE)

## Операторы

управляющие конструкции

операторы SQL, кроме служебных

## Обработка исключительных ситуаций

4

Операторы языка PL/pgSQL объединяются в блоки. В структуре блока можно выделить:

- Необязательную метку, которую можно использовать для устранения неоднозначности в именовании.
- Необязательную секцию, предназначенную для *объявления* локальных переменных и курсоров. В качестве типа можно указать любой тип, определенный в SQL. Можно также сослаться на тип столбца таблицы с помощью конструкции %TYPE.
- Основную секцию исполнения, в которой располагаются *операторы*.
- Необязательную секцию *обработки исключительных ситуаций*.

В качестве операторов можно использовать как команды PL/pgSQL, так и большинство команд SQL, то есть этих два языка интегрированы практически бесшовно. Нельзя использовать служебные команды SQL, такие, как VACUUM. А команды управления транзакциями (например, COMMIT, ROLLBACK) допускаются только в процедурах.

В качестве оператора может выступать и другой (вложенный) PL/pgSQL-блок.

<https://postgrespro.ru/docs/postgresql/16/plpgsql-structure>

<https://postgrespro.ru/docs/postgresql/16/plpgsql-declarations#PLPGSQL-DECLARATION-TYPE>

## Разовое выполнение процедурного кода

- без создания хранимой подпрограммы
- без возможности передать параметры
- без возможности вернуть значение

## Оператор DO языка SQL

Для использования PL/pgSQL не обязательно создавать подпрограммы. Код на PL/pgSQL можно оформить и выполнить как анонимный блок, при помощи SQL-команды DO.

Эту команду можно использовать с разными серверными языками, но если язык не указать явно, то будет использоваться PL/pgSQL.

Код анонимного блока не сохраняется на сервере. Также нет возможности передать в анонимный блок параметры или вернуть из него значение. Хотя косвенно это можно сделать, например, через таблицы.

<https://postgrespro.ru/docs/postgresql/16/sql-do>

## Анонимные блоки

Общая структура блока PL/pgSQL:

```
<<метка>>
DECLARE
    -- объявления переменных
BEGIN
    -- операторы
EXCEPTION
    -- обработка ошибок
END метка;
```

- Все секции, кроме секции операторов, являются необязательными.

---

Минимальный блок PL/pgSQL-кода:

```
=> DO $$
BEGIN
    -- сами операторы могут и отсутствовать
END
$$;

DO
```

Вариант программы Hello, World!

```
=> DO $$
DECLARE
    -- Это однострочный комментарий.
    /* А это — многострочный.
       После каждого объявления ставится знак ';' '.
       Этот же знак ставится после каждого оператора.
    */
    foo text;
    bar text := 'World'; -- также допускается = или DEFAULT
BEGIN
    foo := 'Hello'; -- это присваивание
    RAISE NOTICE '%, %!', foo, bar; -- вывод сообщения
END
$$;

NOTICE: Hello, World!
DO
```

- После BEGIN точка с запятой не ставится!

---

Переменные могут иметь модификаторы:

- CONSTANT — значение переменной не должно изменяться после инициализации;
- NOT NULL — не допускается неопределенное значение.

```
=> DO $$
DECLARE
    foo integer NOT NULL := 0;
    bar CONSTANT text := 42;
BEGIN
    bar := bar + 1; -- ошибка
END
$$;

ERROR: variable "bar" is declared CONSTANT
LINE 6:     bar := bar + 1; -- ошибка
          ^
```

---

Пример вложенных блоков. Переменная во внутреннем блоке перекрывает переменную из внешнего блока, но с помощью меток можно обратиться к любой из них:

```
=> DO $$
<<outer_block>>
DECLARE
    foo text := 'Hello';
BEGIN
    <<inner_block>>
    DECLARE
        foo text := 'World';
    BEGIN
        RAISE NOTICE '%, %!', outer_block.foo, inner_block.foo;
        RAISE NOTICE 'Без метки – внутренняя переменная: %', foo;
    END inner_block;
END outer_block
$$;
```

NOTICE: Hello, World!

NOTICE: Без метки – внутренняя переменная: World

DO

Заголовок подпрограммы не зависит от языка

имя, входные и выходные параметры

для функций: возвращаемое значение, категория изменчивости

Указание LANGUAGE plpgsql

Возврат значений

оператор RETURN

присвоение значений выходным (INOUT, OUT) параметрам

Мы уже познакомились с хранимыми функциями и процедурами на примере языка SQL. Большинство рассмотренных вопросов, связанных с созданием и управлением подпрограммами, относится и к подпрограммам на PL/pgSQL:

- создание, изменение, удаление подпрограмм;
- расположение в системном каталоге (pg\_proc);
- параметры;
- возвращаемое значение и категории изменчивости (для функций);
- перегрузка и полиморфизм;
- и т. д.

Если в SQL подпрограмма возвращала значение, выданное последним SQL-оператором, то в подпрограммах на PL/pgSQL требуется либо присваивать возвращаемые значения формальным INOUT- и OUT-параметрам, либо (для функций) использовать специальный оператор RETURN.



## Подпрограммы PL/pgSQL

Пример функции, возвращающей значение с помощью оператора RETURN:

```
=> CREATE FUNCTION sqr_in(IN a numeric) RETURNS numeric
AS $$
BEGIN
    RETURN a * a;
END
$$ LANGUAGE plpgsql IMMUTABLE;

CREATE FUNCTION
```

Та же функция, но с OUT-параметром. Возвращаемое значение присваивается параметру:

```
=> CREATE FUNCTION sqr_out(IN a numeric, OUT retval numeric)
AS $$
BEGIN
    retval := a * a;
END
$$ LANGUAGE plpgsql IMMUTABLE;

CREATE FUNCTION
```

Та же функция, но с INOUT-параметром. Такой параметр используется и для принятия входного значения, и для возврата значения функции:

```
=> CREATE FUNCTION sqr_inout(INOUT a numeric)
AS $$
BEGIN
    a := a * a;
END
$$ LANGUAGE plpgsql IMMUTABLE;

CREATE FUNCTION
```

```
=> SELECT sqr_in(3), sqr_out(3), sqr_inout(3);
```

| sqr_in |  | sqr_out |  | sqr_inout |
|--------|--|---------|--|-----------|
| 9      |  | 9       |  | 9         |

(1 row)

## IF

стандартный условный оператор

## CASE

похож на CASE языка SQL, но не возвращает значение

## Внимание: трехзначная логика

условие должно быть истинно; false и NULL не подходят

PL/pgSQL предлагает два условных оператора: IF и CASE.

Первый — совершенно стандартный оператор, имеющийся во всех языках.

Второй похож на конструкцию CASE в языке SQL, но это именно оператор, он не возвращает значение. Это некий аналог оператора switch в C или Java.

Важно всегда помнить о том, что логические выражения в SQL (и, следовательно, в PL/pgSQL) могут принимать *три* значения: true, false и NULL. Условие срабатывает, только когда оно истинно, и не срабатывает, когда ложно *или не определено*. Это одинаково верно и для условий WHERE в SQL, и для условных операторов PL/pgSQL.

<https://postgrespro.ru/docs/postgresql/16/plpgsql-control-structures#PLPGSQL-CONDITIONALS>

## Условные операторы

Общий вид оператора IF:

```
IF условие THEN
  -- операторы
ELSIF условие THEN
  -- операторы
ELSE
  -- операторы
END IF;
```

- Секция ELSIF может повторяться несколько раз, а может отсутствовать.
- Секция ELSE может отсутствовать.
- Выполняются операторы, соответствующие первому истинному условию.
- Если ни одно из условий не истинно, выполняются операторы ELSE (если есть).

---

Пример функции, использующей условный оператор для форматирования номера телефона. Функция возвращает значение составного типа:

```
=> CREATE FUNCTION fmt (IN phone text, OUT code text, OUT num text)
AS $$
BEGIN
  IF phone ~ '^[0-9]*$' AND length(phone) = 10 THEN
    code := substr(phone,1,3);
    num  := substr(phone,4);
  ELSE
    code := NULL;
    num  := NULL;
  END IF;
END
$$ LANGUAGE plpgsql IMMUTABLE;

CREATE FUNCTION

=> SELECT fmt('8122128506');

      fmt
-----
(812,2128506)
(1 row)
```

---

Общий вид оператора CASE (первый вариант — по условию):

```
CASE
  WHEN условие THEN
    -- операторы
  ELSE
    -- операторы
END CASE;
```

- Секция WHEN может повторяться несколько раз.
- Секция ELSE может отсутствовать.
- Выполняются операторы, соответствующие первому истинному условию.
- Если ни одно из условий не истинно, выполняются операторы ELSE (отсутствие ELSE в таком случае — ошибка).

---

Пример использования:

```

=> DO $$
DECLARE
    code text := (fmt('8122128506')).code;
BEGIN
    CASE
        WHEN code IN ('495','499') THEN
            RAISE NOTICE '% — Москва', code;
        WHEN code = '812' THEN
            RAISE NOTICE '% — Санкт-Петербург', code;
        WHEN code = '384' THEN
            RAISE NOTICE '% — Кемеровская область', code;
        ELSE
            RAISE NOTICE '% — Прочие', code;
    END CASE;
END
$$;

NOTICE:  812 — Санкт-Петербург
DO

```

---

Общий вид оператора CASE (второй вариант — по выражению):

```

CASE выражение
    WHEN значение, ... THEN
        -- операторы
    ELSE
        -- операторы
END CASE;

```

- Секция WHEN может повторяться несколько раз.
- Секция ELSE может отсутствовать.
- Выполняются операторы, соответствующие первому истинному условию «выражение = значение».
- Если ни одно из условий не истинно, выполняются операторы ELSE (отсутствие ELSE в таком случае — ошибка).

---

При однотипных условиях эта форма CASE может оказаться компактней:

```

=> DO $$
DECLARE
    code text := (fmt('8122128506')).code;
BEGIN
    CASE code
        WHEN '495', '499' THEN
            RAISE NOTICE '% — Москва', code;
        WHEN '812' THEN
            RAISE NOTICE '% — Санкт-Петербург', code;
        WHEN '384' THEN
            RAISE NOTICE '% — Кемеровская область', code;
        ELSE
            RAISE NOTICE '% — Прочие', code;
    END CASE;
END
$$;

NOTICE:  812 — Санкт-Петербург
DO

```

Цикл FOR по диапазону чисел

Цикл WHILE с предусловием

Бесконечный цикл

Цикл может иметь метку, как блок

Управление

выход из цикла (EXIT)

переход на новую итерацию (CONTINUE)

Для повторного выполнения набора операторов PL/pgSQL предлагает несколько видов циклов:

- цикл FOR по диапазону целых чисел;
- цикл WHILE с предусловием;
- бесконечный цикл.

Цикл представляет собой разновидность блока и может иметь собственную метку.

Выполнением цикла можно дополнительно управлять, иницилируя переход на новую итерацию или выход из цикла.

<https://postgrespro.ru/docs/postgresql/16/plpgsql-control-structures#PLPGSQL-CONTROL-STRUCTURES-LOOPS>

Цикл FOR может работать не только по диапазону чисел, но и по результатам выполнения запроса и по массивам. Эти варианты цикла FOR будут рассмотрены в следующих темах.

## Циклы

В PL/pgSQL все циклы используют общую конструкцию:

```
LOOP
    -- операторы
END LOOP;
```

К ней может добавляться заголовок, определяющий условие выхода из цикла.

---

Цикл по диапазону FOR повторяется, пока счетчик цикла пробегает значения от нижней границы до верхней. С каждой итерацией счетчик увеличивается на 1 (но инкремент можно изменить в необязательной фразе BY).

```
FOR имя IN низ .. верх BY инкремент
LOOP
    -- операторы
END LOOP;
```

- Переменная, выступающая счетчиком цикла, объявляется неявно и существует только внутри блока LOOP — END LOOP.
- 

При указании REVERSE значение счетчика на каждой итерации уменьшается, а нижнюю и верхнюю границы цикла нужно поменять местами:

```
FOR имя IN REVERSE верх .. низ BY инкремент
LOOP
    -- операторы
END LOOP;
```

---

Пример использования цикла FOR — функция, "переворачивающая" строку:

```
=> CREATE FUNCTION reverse_for (line text) RETURNS text
AS $$
DECLARE
    line_length CONSTANT int := length(line);
    retval text := '';
BEGIN
    FOR i IN 1 .. line_length
    LOOP
        retval := substr(line, i, 1) || retval;
    END LOOP;
    RETURN retval;
END
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

CREATE FUNCTION

Напомним, что строгая (STRICT) функция немедленно возвращает NULL, если хотя бы один из входных параметров не определен. Тело функции при этом не выполняется.

---

Цикл WHILE выполняется до тех пор, пока истинно условие:

```
WHILE условие
LOOP
    -- операторы
END LOOP;
```

---

Та же функция, обращающая строку, с использованием цикла WHILE:

```
=> CREATE FUNCTION reverse_while (line text) RETURNS text
AS $$
DECLARE
    line_length CONSTANT int := length(line);
    i int := 1;
    retval text := '';
BEGIN
    WHILE i <= line_length
    LOOP
        retval := substr(line, i, 1) || retval;
        i := i + 1;
    END LOOP;
    RETURN retval;
END
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

## CREATE FUNCTION

Цикл LOOP без заголовка выполняется бесконечно. Для выхода используется оператор EXIT:

**EXIT** метка **WHEN** условие;

- Метка необязательна; если не указана, будет прерван самый вложенный цикл.
- Фраза WHEN также необязательна; при отсутствии цикл прерывается безусловно.

Пример использования цикла LOOP:

```
=> CREATE FUNCTION reverse_loop (line text) RETURNS text
AS $$
DECLARE
    line_length CONSTANT int := length(reverse_loop.line);
    i int := 1;
    retval text := '';
BEGIN
    <<main_loop>>
    LOOP
        EXIT main_loop WHEN i > line_length;
        retval := substr(reverse_loop.line, i,1) || retval;
        i := i + 1;
    END LOOP;
    RETURN retval;
END
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

## CREATE FUNCTION

- Тело функции помещается в неявный блок, метка которого совпадает с именем функции. Поэтому к параметрам можно обращаться как «имя\_функции.параметр».

Убедимся, что все функции работают правильно:

```
=> SELECT reverse_for('главрыба') as "for",
        reverse_while('главрыба') as "while",
        reverse_loop('главрыба') as "loop";

   for   |  while   |   loop
-----+-----+-----
 абырвалг | абырвалг | абырвалг
(1 row)
```

Замечание. В PostgreSQL есть встроенная функция reverse.

Иногда бывает полезен оператор CONTINUE, начинающий новую итерацию цикла:

```
=> DO $$
DECLARE
    s integer := 0;
BEGIN
    FOR i IN 1 .. 100
    LOOP
        s := s + i;
        CONTINUE WHEN mod(i, 10) != 0;
        RAISE NOTICE 'i = %, s = %', i, s;
    END LOOP;
END
$$;
```

```
NOTICE: i = 10, s = 55
NOTICE: i = 20, s = 210
NOTICE: i = 30, s = 465
NOTICE: i = 40, s = 820
NOTICE: i = 50, s = 1275
NOTICE: i = 60, s = 1830
NOTICE: i = 70, s = 2485
NOTICE: i = 80, s = 3240
NOTICE: i = 90, s = 4095
NOTICE: i = 100, s = 5050
DO
```

## Любое выражение вычисляется в контексте SQL

- выражение автоматически преобразуется в запрос
- запрос подготавливается
- переменные PL/pgSQL подставляются как параметры

## Особенности

- можно использовать все возможности SQL, включая подзапросы
- невысокая скорость выполнения,
- хотя разобранный запрос (и, возможно, план запроса) кешируются
- неоднозначности при разрешении имен требуют внимания

13

Выражения, встречающиеся в PL/pgSQL-коде, вычисляются с помощью SQL-запросов к базе данных. Интерпретатор сам строит нужный запрос, формируя оператор вида `SELECT <expr>` и заменяя переменные PL/pgSQL на параметры, затем подготавливает его (при этом разобранный запрос кешируется, как это обычно и происходит с подготовленными операторами), после чего планирует и выполняет.

С одной стороны, такой подход не способствует скорости работы PL/pgSQL, зато обеспечивает теснейшую интеграцию с SQL. Фактически, в выражениях можно использовать любые возможности SQL без ограничений, включая вызов встроенных и пользовательских функций, выполнение подзапросов и т. п.

Начиная с версии PostgreSQL 14 оптимизировано выполнение простых выражений (как минимум не содержащих обращений к таблицам) — такие выражения обрабатываются синтаксическим анализатором сервера напрямую, без обращений к планировщику.

<https://postgrespro.ru/docs/postgresql/16/plpgsql-expressions>



## Вычисление выражений

Выражения в PL/pgSQL выполняются SQL-движком. Таким образом, в PL/pgSQL доступно ровно то, что доступно в SQL. Например, если в SQL можно использовать конструкцию CASE, то точно такая же конструкция будет работать и в коде на PL/pgSQL (в качестве выражения; не путайте с оператором CASE ... END CASE, который есть только в PL/pgSQL):

```
=> DO $$
BEGIN
    RAISE NOTICE '%', CASE 2+2 WHEN 4 THEN 'Все в порядке' END;
END
$$;

NOTICE: Все в порядке
DO
```

В выражениях можно использовать и подзапросы:

```
=> DO $$
BEGIN
    RAISE NOTICE '%', (
        SELECT code
        FROM (VALUES (1, 'Раз'), (2, 'Два')) t(id, code)
        WHERE id = 1
    );
END
$$;

NOTICE: Раз
DO
```

Еще пример с вычислением выражения в PL/pgSQL — сколько функций «обращения» строк у нас в результате получилось?

```
=> DO $$
DECLARE
    s integer;
BEGIN
    s := count(*) FROM pg_proc WHERE proname LIKE 'reverse%';
    RAISE NOTICE 'Всего функций "reverse" : %', s;
END
$$;

NOTICE: Всего функций "reverse" : 4
DO
```

PL/pgSQL — доступный по умолчанию, интегрированный с SQL, удобный и простой в использовании язык

Управление подпрограммами на PL/pgSQL не отличается от работы с подпрограммами на других языках

DO — команда SQL для выполнения анонимного блока

Переменные PL/pgSQL могут использовать любые типы SQL

Язык поддерживает обычные управляющие конструкции, такие как условные операторы и циклы



1. Измените функцию `book_name` так, чтобы длина возвращаемого значения не превышала 45 символов. Если название книги при этом обрезается, оно должно завершаться на троеточие.  
Проверьте реализацию в SQL и в приложении; при необходимости добавьте книг с длинными названиями.
2. Снова измените функцию `book_name` так, чтобы избыточно длинное название обрезалось по границе целого слова (укорачивание по словам).  
Проверьте реализацию.

## 1. Например:

Путешествия в некоторые удалённые страны мира в четырёх частях: сочинение Лемюэля Гулливера, сначала хирурга, а затем капитана нескольких кораблей →

→ Путешествия в некоторые удалённые страны м...

Вот некоторые случаи, которые имеет смысл проверить:

- длина названия меньше 45 символов (не должно измениться);
- длина названия ровно 45 символов (не должно измениться);
- длина названия 46 символов (от названия должны быть отрезаны 4 символа, т. к. добавятся еще три точки).

Лучше всего написать и отладить отдельную функцию укорачивания, которую затем использовать в `book_name`. Это полезно и по другим соображениям:

- такая функция может пригодиться где-то еще;
- каждая функция будет выполнять ровно одну задачу.

## 2. Например:

Путешествия в некоторые удалённые страны мира в четырёх частях: сочинение Лемюэля Гулливера, сначала хирурга, а затем капитана нескольких кораблей →

→ Путешествия в некоторые удалённые страны...

Как поведет себя ваша реализация, если название состоит из одного длинного слова без пробелов?

## 1. Укорачивание названия книги

Напишем более универсальную функцию, принимающую строку, максимальную длину и суффикс, добавляемый при укорачивании. Это не потребует усложнения кода, и позволит обойтись без «магических констант».

```
=> CREATE FUNCTION shorten(
    s text,
    max_len integer DEFAULT 45,
    suffix text DEFAULT '...'
)
RETURNS text AS $$
DECLARE
    suffix_len integer := length(suffix);
BEGIN
    RETURN CASE WHEN length(s) > max_len
        THEN left(s, max_len - suffix_len) || suffix
        ELSE s
    END;
END
$$ IMMUTABLE LANGUAGE plpgsql;

CREATE FUNCTION

Проверим:

=> SELECT shorten(
    'Путешествия в некоторые удаленные страны мира в четырех частях: сочинение Лемюэля Гулливера, сначала хирурга, а затем капитана нескольких кораблей'
);

          shorten
-----
Путешествия в некоторые удаленные страны м...
(1 row)

=> SELECT shorten(
    'Путешествия в некоторые удаленные страны мира в четырех частях: сочинение Лемюэля Гулливера, сначала хирурга, а затем капитана нескольких кораблей',
    30
);

          shorten
-----
Путешествия в некоторые уда...
```

Используем написанную функцию:

```
=> CREATE OR REPLACE FUNCTION book_name(book_id integer, title text)
RETURNS text
STABLE LANGUAGE sql
BEGIN ATOMIC
SELECT shorten(book_name.title) ||
    CASE WHEN (right(shorten(book_name.title), 3) != '...') THEN ' '::text ELSE ' ' END ||
    string_agg(
        author_name(a.last_name, a.first_name, a.middle_name), ', '
        ORDER BY ash.seq_num
    )
FROM authors a
JOIN authorship ash ON a.author_id = ash.author_id
WHERE ash.book_id = book_name.book_id;
END;

CREATE FUNCTION
```

## 2. Укорачивание названия книги с переносом по словам

```
=> CREATE OR REPLACE FUNCTION shorten(
    s text,
    max_len integer DEFAULT 45,
    suffix text DEFAULT '...'
)
RETURNS text
AS $$
DECLARE
    suffix_len integer := length(suffix);
    short text := suffix;
BEGIN
    IF length(s) < max_len THEN
        RETURN s;
    END IF;
    FOR pos in 1 .. least(max_len-suffix_len+1, length(s))
    LOOP
        IF substr(s,pos-1,1) != ' ' AND substr(s,pos,1) = ' ' THEN
            short := left(s, pos-1) || suffix;
            END IF;
        END LOOP;
    RETURN short;
END
$$ IMMUTABLE LANGUAGE plpgsql;

CREATE FUNCTION

Проверим:

=> SELECT shorten(
    'Путешествия в некоторые удаленные страны мира в четырех частях: сочинение Лемюэля Гулливера, сначала хирурга, а затем капитана нескольких кораблей'
);

          shorten
-----
Путешествия в некоторые удаленные страны...
```

```
=> SELECT shorten(
  'Путешествия в некоторые удаленные страны мира в четырех частях: сочинение Лемюэля Гулливера, сначала хирурга, а затем капитана нескольких кораблей',
  30
);

      shorten
-----
Путешествия в некоторые...
(1 row)
```

1. Напишите PL/pgSQL-функцию, которая возвращает строку заданной длины из случайных символов.
2. Задача про игру в «наперстки».

В одном из трех наперстков скрыт выигрыш.

Игрок выбирает один из этих трех. Ведущий убирает один из двух оставшихся наперстков (обязательно пустой) и дает игроку возможность поменять решение, то есть выбрать второй из двух оставшихся.

Есть ли смысл игроку менять выбор или нет смысла менять первоначальный вариант?

Задание: используя PL/pgSQL, посчитайте вероятность выигрыша и для начального выбора, и для измененного.

17

Предварительно можно создать функцию `rnd_integer`, которая возвращает случайное целое число в заданном диапазоне. Функция будет полезна для решения обоих заданий.

Например: `rnd_integer(30, 1000) → 616`

1. Помимо длины строки на вход функции можно подавать список допустимых символов. По умолчанию, это могут быть все символы алфавита, числа и некоторые знаки. Для определения случайных символов из списка можно использовать функцию `rnd_integer`. Объявление функции может быть таким:

```
CREATE FUNCTION rnd_text(
    len int,
    list_of_chars text DEFAULT
    'АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдеёжзийклмнопрстуфхцчшщъыьэю
    яАВСDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_0123456789 '
) RETURNS text AS ...
```

Пример вызова: `rnd_text(10) → 'ЛждфбЁ_00J'`

2. Для решения можно использовать анонимный блок.

Сначала нужно реализовать одну игру и посмотреть, какой вариант выиграл: начальный или измененный. Для загадывания и угадывания одного из трех наперстков можно использовать `rnd_integer(1,3)`.

Затем игру поместить в цикл и «сыграть», например, 1000 раз, подсчитывая, какой вариант сколько раз победил. В конце через `RAISE NOTICE` вывести значения счетчиков и выявить победивший вариант (или отсутствие такового).

## 1. Случайная строка заданного размера

```
=> CREATE DATABASE plpgsql_introduction;
```

```
CREATE DATABASE
```

```
=> \c plpgsql_introduction
```

You are now connected to database "plpgsql\_introduction" as user "student".

Вначале определим вспомогательную функцию для получения случайного целого числа в заданном диапазоне. Такую функцию легко написать на чистом SQL, но здесь представлен вариант на PL/pgSQL:

```
=> CREATE FUNCTION rnd_integer(min_value integer, max_value integer)
RETURNS integer
AS $$
DECLARE
    retval integer;
BEGIN
    IF max_value <= min_value THEN
        RETURN NULL;
    END IF;

    retval := floor(
        (max_value+1 - min_value)*random()
    )::integer + min_value;
    RETURN retval;
END
$$ STRICT LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Проверяем работу:

```
=> SELECT rnd_integer(0,1) as "0 - 1",
        rnd_integer(1,365) as "1 - 365",
        rnd_integer(-30,30) as "-30 - +30"
FROM generate_series(1,10);
```

| 0 - 1 | 1 - 365 | -30 - +30 |
|-------|---------|-----------|
| 1     | 213     | 1         |
| 0     | 200     | -3        |
| 1     | 148     | -14       |
| 1     | 232     | -6        |
| 0     | 294     | -6        |
| 1     | 228     | -12       |
| 1     | 313     | 6         |
| 0     | 86      | -20       |
| 1     | 161     | 18        |
| 0     | 215     | -13       |

(10 rows)

Функция гарантирует равномерное распределение случайных значений по всему диапазону, включая граничные значения:

```
=> SELECT rnd_value, count(*)
FROM (
    SELECT rnd_integer(1,5) AS rnd_value
    FROM generate_series(1,100_000)
)
GROUP BY rnd_value ORDER BY rnd_value;
```

| rnd_value | count |
|-----------|-------|
| 1         | 19903 |
| 2         | 20005 |
| 3         | 20290 |
| 4         | 19866 |
| 5         | 19936 |

(5 rows)

Теперь можно приступить к написанию функции для получения случайной строки заданного размера. Будем использовать функцию rnd\_integer для получения случайного символа из списка.

```
=> CREATE FUNCTION rnd_text(
    len int,
    list_of_chars text DEFAULT 'АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдеёжзийклмнопрстуфхцчшщъыьэюяABCDEFGHIJKLMNOPQRSTUVWXYZ_0123456789'
) RETURNS text
AS $$
DECLARE
    len_of_list CONSTANT integer := length(list_of_chars);
    i integer;
    retval text := '';
BEGIN
    FOR i IN 1 .. len
    LOOP
        -- добавляем к строке случайный символ
        retval := retval ||
            substr(list_of_chars, rnd_integer(1,len_of_list),1);
    END LOOP;
    RETURN retval;
END
$$ STRICT LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Проверяем:

```
=> SELECT rnd_text(rnd_integer(1,30)) FROM generate_series(1,10);
```

```

        rnd_text
-----
УыйZv6HEIwЦB2эbЕФЬРсиФЧovJл
ынЕССБйнGqРiЕЯМчЭvLwэz5EVrЖг
в0ФхSo
ЕпыY0lPлPvU75oK20g
Е0ёЯLN_пNmG_эFПефиXKlqеижPB
Ь
ЫlШ2ьвлC
лрBT1RKИе0биЕ
ХоЛжЙНшьПпCх8EgvСИ05pЛЖс9FГsoX
гЛМЕiэжщИQМБКм0КтPvxг3E
(10 rows)

```

## 2. Игра в наперстки

Для загадывания и угадывания наперстка используем rnd\_integer(1,3).

```

=> DO $$
DECLARE
    x integer;
    choice integer;
    new_choice integer;
    remove integer;
    total_games integer := 1000;
    old_choice_win_counter integer := 0;
    new_choice_win_counter integer := 0;
BEGIN
    FOR i IN 1 .. total_games
    LOOP
        -- Загадываем выигрышный наперсток
        x := rnd_integer(1,3);

        -- Игрок делает выбор
        choice := rnd_integer(1,3);

        -- Убираем один неверный ответ, кроме выбора игрока
        FOR i IN 1 .. 3
        LOOP
            IF i NOT IN (x, choice) THEN
                remove := i;
                EXIT;
            END IF;
        END LOOP;

        -- Нужно ли игроку менять свой выбор?
        -- Что лучше: оставить choice или заменить его на оставшийся?

        -- Измененный выбор
        FOR i IN 1 .. 3
        LOOP
            IF i NOT IN (remove, choice) THEN
                new_choice := i;
                EXIT;
            END IF;
        END LOOP;

        -- Или начальный, или новый выбор обязательно выиграют
        IF choice = x THEN
            old_choice_win_counter := old_choice_win_counter + 1;
        ELSIF new_choice = x THEN
            new_choice_win_counter := new_choice_win_counter + 1;
        END IF;
    END LOOP;

    RAISE NOTICE 'Выиграл начальный выбор: % из %',
        old_choice_win_counter, total_games;
    RAISE NOTICE 'Выиграл измененный выбор: % из %',
        new_choice_win_counter, total_games;
END
$$;

NOTICE:  Выиграл начальный выбор: 329 из 1000
NOTICE:  Выиграл измененный выбор: 671 из 1000
DO

```

Вначале мы выбираем 1 из 3, поэтому вероятность начального выбора 1/3. Если же выбор изменить, то изменится и вероятность на противоположные 2/3.

Таким образом, вероятность выиграть при смене выбора выше. Поэтому есть смысл выбор поменять.

```

=> \c postgres

```

You are now connected to database "postgres" as user "student".

```

=> DROP DATABASE plpgsql_introduction;

```

```

DROP DATABASE

```